Microcoding an abstract machine for parallel logic programming

A. Rizk, J. Garcia

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 1150

Programme 1
Programmation, Calcul Symbolique
et Intelligence Artificielle

MICROCODING AN ABSTRACT MACHINE FOR PARALLEL LOGIC PROGRAMMING

Antoine RIZK
José GARCIA

Décembre 1989

RR.1150

Programme 1

Microcoding an Abstract Machine for Parallel Logic Programming

A. RIZK
INRIA
Domaine de Voluceau - Rocquencourt
B.P. 105
78153 LE CHESNAY Cedex
☎ : (1) 39 63 52 38

J. GARCIA
Bull
68, route de Versailles
78430 LOUVECIENNES

Abstract

This paper shows the advantages of implementing an abstract intermediate machine for a parallel logical language, on the lower hardware level - the FIRMWARE level - of a physical machine which implements basic hardware mechanisms for fast symbolic computation.

μSyC, which is a μprogrammable Symbolic Coprocessor under development at the Bull Research Center, has been chosen as a target architecture and the abstract machine in question is the Sequential Parlog Machine which is based on the AND/OR tree execution model for concurrent logical languages such as PARLOG, CP and GHC.

The paper describes the μSyC architecture, the language PARLOG, the AND/OR tree model, and the mapping of this model on μSyC. Finally, results for a series of classical benchmark tests is given and compared to other available implementations.

Keywords : PARLOG, μSyc, Logic Programming, Language implementation, microcode.

Une Implémentation de PARLOG en Microcode

Résumé

Nous décrivons dans cet article une implémentation du langage logique concurrent PARLOG, sur le coprocesseur symbolique microprogrammable μSyC de BULL.

Après une présentation de PARLOG et μSyc, nous décrivons la machine intermédiaire qui a été réalisée en microcode. Nous présentons des tests de performance qui ont été effectués sur cette machine.

Mots-clés : PARLOG, μSyc, Programmation en Logique, Implémentation de Langage, microcode.

I Introduction

The PARLOG Scenario

The study described in this paper has been carried out in the context of a European ESPRIT project that aims to study advanced components of future workstations.

One of the scenarii proposed for a future workstation is based around PARLOG, a parallel logical language developed at Imperial College (UK). In such a machine, PARLOG will be located at the bottom software layer in firmware, with other software layers, including the operating system, built on top (Fig. I.1).

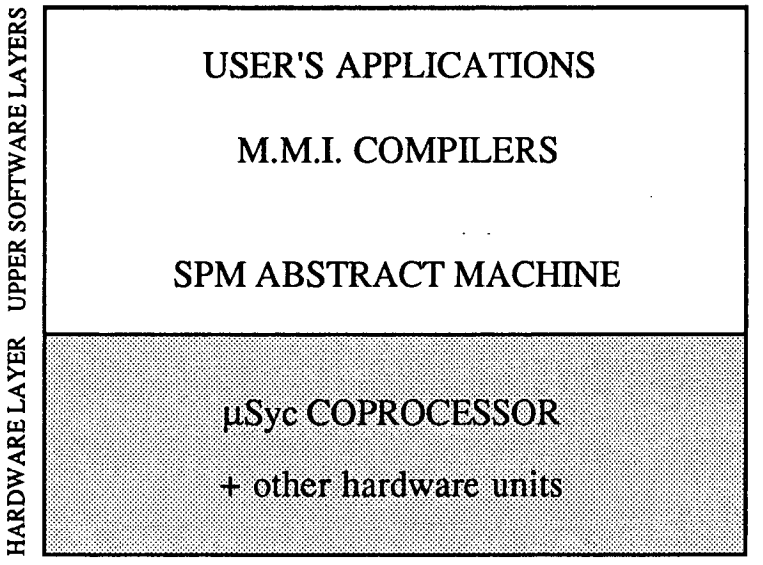


Fig. I.1

The reasons for choosing a concurrent logical language as the base component of a computer system have been amply discussed in [Sha 87] and the proponents of the Japanese fifth generation programme. Putting PARLOG at the firmware level further provides the required fast logic support for future AI applications.

Implementing a language by building a dedicated machine and microprogramming in firmware the whole or part of its basic instructions is not an innovative choice. Indeed, over the past few years we have seen a series of successful examples : the Explorer Lisp machine from Texas, the Prolog machine PSI from ICOT, the LMI machines, Symbolics machine, the MAIA machine, the Prolog machine KCM from ECRC. In addition, all these architectures often use other features like tag checking mechanism, pipelining and caching. However, our attempt is the first of the kind for a parallel logical language : a powerful formalism that badly needs a performance boost especially on single processor architectures.

The next sections will expose briefly the architecture of µSyC followed by a presentation of the AND/OR tree computational model and more concretely as it is realised in the Sequential PARLOG Machine (SPM). We will then show how the execution of the SPM abstract machine can be efficiently improved by microprogramming its instructions in the µSyC control store. We do not claim that µSyC is the ideal architecture for supporting PARLOG, it has however proven very effective for mapping the SPM which is adequately designed for running on a uniprocessor machine.

Finally, we will provide the performance obtained with μSyC for several classical benchmarks and compare them to other implementations.

II μSyC architecture

μSyC [Cou 87, Gon 84, Gon 86] is a microprogrammable symbolic microprocessor which may be seen by the host processor (eg. the 68020) as a DMA-type peripheral, i.e. it may request the mastership of the bus. Its external microprogram memory is accessed via a 12-bit micro-address bus and a 69-bit micro-instruction bus (fig. II-1).

As shown in fig. II-2, μSyC is classically composed of a data path, a micro-instruction sequencer, a memory controller, and host CPU interface logic and registers. As μSyC is essentially a prototype, the firmware must be easy to modify. For this reason its microprogram memory is still not integrated on chip.

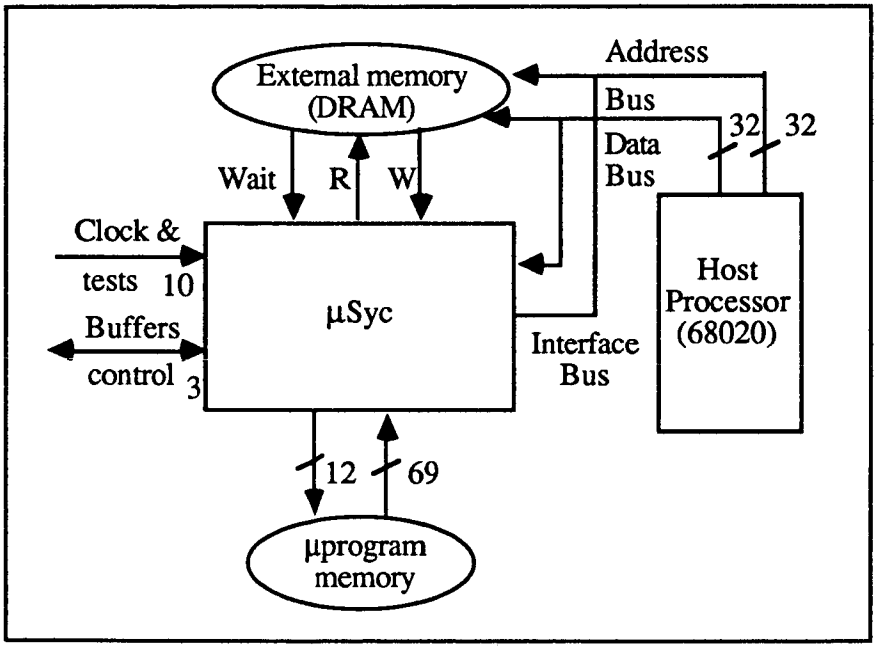


Fig II-1 : μSyc external overview

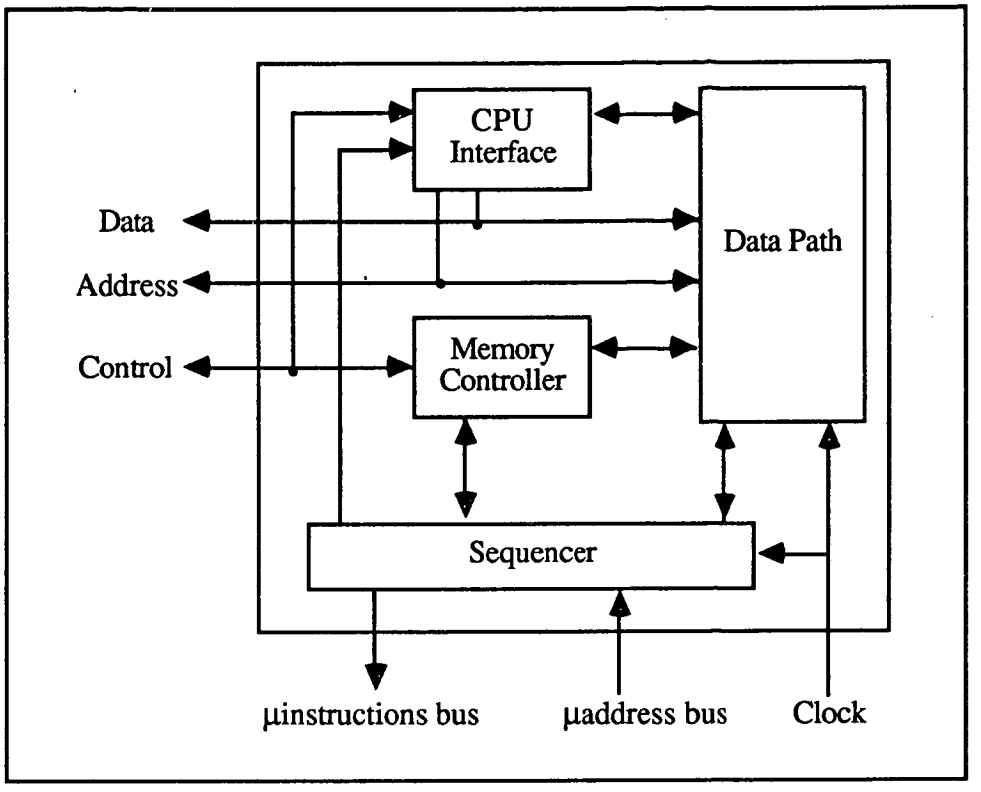


Fig II-2 : µSyC internal organization

II.1 The data path

The data path is organized around two main busses, the data bus and the address bus. There are in addition smaller secondary busses for the connection between the main busses and independent operators like counters and byte extractors.

The data bus connects together sixteen 32-bit registers, one ALU, one shifter, a byte injector, a bit tester that can test a bit among 32, and a set of four wired 32-bit constants.

The address bus connects together one ALU, sixteen 32-bit registers and four 32-bit wired constants. In addition, there is a tag comparator and a four bit three state gate that allows a tag to be associated to a 28-bit pointer.

The other operators are two byte extractors, one byte comparator and two sets of counters. Each counter set is made up of four registers and a decrementer. These counters can drive directly the byte extractors and the byte injector, giving the position in which the byte is to be extracted from or inserted into a 32-bit word.

II.2 The sequencer

The sequencer of µSyC contains an eight 12-bit address stack for sub-microprogram calls, an 8-bit counter for short loops, and an address calculator able to perform the special multi-destination branches and calls mentioned in the following subsections.

II.3 The memory controller

The memory controller allows the microprogrammer to ignore such details as the number of cycles the memory needs in order to perform a read operation, wait states, bus errors, etc.

If the memory controller is asked to access the memory, and if for any reason the memory is not accessible at this time (for example, a previous read or write operation is not yet finished), the memory controller sets the HALT signal. This signal is received by the sequencer and by the data path, and both of them freeze their current state until the HALT signal is released by the memory controller.

Nevertheless, if the microprogrammer knows the memory access time, he can take advantage of the micro-parallelism of µSyC by interleaving, when possible, read/write operations and internal operations. As a matter of fact, it is possible, thanks to the 69-bit micro-instruction word, to command 1 to 8 simultaneous actions, e.g. two arithmetic operations, two byte extractions with the decrement of associated counters, a byte comparison and a memory access request.

This is an obvious advantage of µSyC over a conventional microprocessor, for the latter cannot do anything but wait during the memory wait states.

II.4 The host CPU interface

The host CPU and µSyC communicate via 9 interface registers and 9 control signals. All of the interface registers are read/write for both µSyC and the host CPU.

One of the registers serves as a status word, and a particular bit in this word is used to start µSyC at the beginning. After a reset, µSyC does some initializations and starts looping until this bit is set by the host CPU.

II.5 Specialized operators in µSyC

µSyC has specialized hardware features in order to deal efficiently with character strings and tagged pointers.

It has a 32-bit data bus and a 32 bit-address bus. In order to minimize the number of memory accesses, µSyC only reads or writes 32-bit words, but it has specialized operators able to extract one byte from a 32-bit word, or insert a byte into a 32-bit word, at any 8-bit aligned position.

There are two byte extractors (with their associated registers) and one byte comparator, that can operate in parallel. These features make character string operations - especially comparison - very efficient in µSyC. One byte injector is used to prepare the results before writing them - 32 bits at a time - back into memory.

µSyC also has specialized hardware features that make tag manipulation easy and efficient. Tags are 4 bits long and are located in the 4 higher-order bits of a 32 bit word. Thus, 16 different identifier types can be associated with pointers, while 28 bits remain available to address a 256 megabyte memory space.

The basic operations that µSyC can easily perform on tags are the following ones :

- insert a tag into the 4 high bits of a 32 bit word ;

- compare the tag in a 32 bit word with a 4 bit value ;

- branch to one address among 16 addresses, depending on the value of a tag (in one cycle).

II.6 Multi-destination branches

First, let us examine how the destination address is calculated from a constant micro-address and a tag. A micro-address is 12 bits long. The destination address is obtained by inserting the tag at the 8th position in the constant micro-address. Thus, if we have a multi-destination branch micro-function with a constant micro-address (in binary) :

a11 a10 a9 a8 a7 a6 a5 a4 a3 a2 a1 a0

and if the current tag is :

t3 t2 t1 t0

then the destination micro-address will be :

a11 a10 a9 a8 t3 t2 t1 t0 a3 a2 a1 a0

Of course, the microprogrammer only deals with symbolic labels, the micro-address calculator of our micro-assembler does all the difficult work.

It is possible to use only a 3, 2, or even a 1 bit tag, avoiding putting useless constraints on dummy micro-addresses. In the above example, using a 2 bit tag would have given :

a11 a10 a9 a8 t3 t2 a5 a4 a3 a2 a1 a0

In all the multi-destination branch micro-functions (there are 4 multi-conditional branches and calls), the tag can be replaced by a set of 1 bit flags. These flags memorize various signals from ALUs (zero, carry), shifter, decrementers or bit tester.

This last feature would hardly be possible without the use of a powerful micro-assembler which assumes the task of allocating micro-addresses to label symbols under some constraints.

III PARLOG - THE AND/OR TREE MODEL, Ros

A detailed description of PARLOG and the SPM can be found in [Gre 87a, 87b, Cla 85, Fos 86]. In this section we will focus our interest on a special computational model called AND/OR tree model which is implemented in the SPM. In this model, PARLOG execution is represented by a tree of processes dynamically constructed, each node of which carries a process descriptor PD, a sequence of primitive instructions and a code pointer attached to it. A process can be in two states : runnable or suspended. It is suspended whenever it is waiting for an offspring process to be finished or for a variable to be bound. The process descriptor contains the type of the process which is used to determine the next process to be scheduled once the offspring processes have terminated.

A process can be of two types :

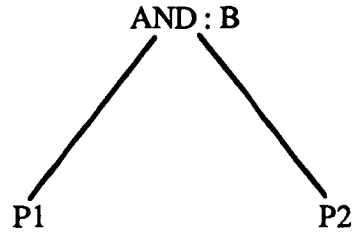
Type "AND" :

Let us assume that the current process's code pointer adresses the following code :

(P1 // P2) & B

where // means that goals P1 and P2 can be solved in parallel
and & is the sequential conjunction implying that (P1//P2) must be solved before B can be solved.

The current process becomes an AND process, its code pointer is set to goal B and two offspring processes are created for goal P1 and P2. Both processes become then RUNNABLE.



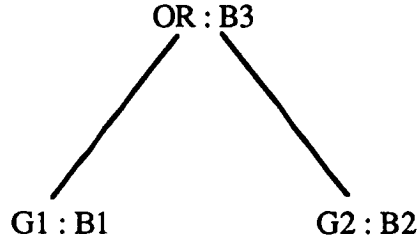
In this case an offspring process succeeds, and is then removed from the process tree. On the other hand, if any of P1 or P2 fails, the parent process automatically fails.

Type "OR" :

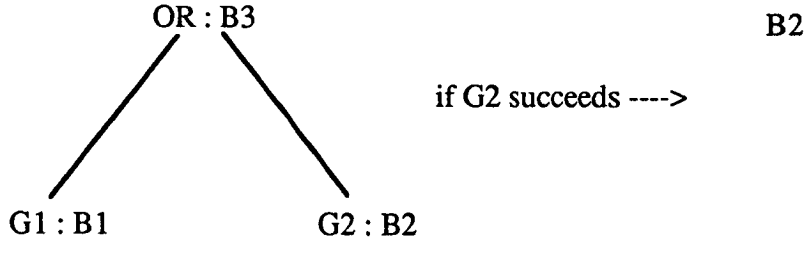
Suppose that we are solving a call to predicate P defined as follows :

```
C1 :    P <- G1: B1 //
C2 :    P <- G2: B2 ;        -- the sequential search operator
C3 :    P <- B3.
```

The current process becomes an OR process, its code pointer is set to B3, and two offspring processes (that represent RUNNABLE processes) are created for clauses C1 and C2.



Clauses C1 and C2 will be tried in parallel. Each clause is composed of a guard G and a body B. Once a guard of a clause succeeds, the remaining sibling processes are removed from the process tree. In our case, if guard G2 succeeds, B2 is resumed at the parent level (by setting its code pointer to B2) as follows :



Clause C3 will be activated only in case both clauses C1 and C2 fail, either because their guards have failed or because their bodies have failed.

IV The SPM on μSyC

SPM is the uniprocessor version of the AND/OR tree model which implements several optimization techniques in order to avoid the overheads incurred by scheduling in a naive AND/OR tree model :

a - Together with the process tree, SPM maintains two other scheduling structures: runnable lists (for runnable processes) and suspension lists (for suspended processes). Only one process is active at any one time.

b - SPM also tries to minimize the profusion of processes by a depth first scheduling strategy.

If we go back to our first example : (P1 // P2) & B

Instead of creating two new processes (one for P1 and another for P2), SPM will only create a new process for P1 and set the current process to the conjunction P2 & B.

(P1 // P2) & B --> RUNNABLE/AND : P2 & B

P1

If P1 suspends, its process is added to the suspension queue and its parent is resumed. A new process is then spawn for P2 and the parent process is set to B.

If P1 succeeds, its process is removed from the process tree and its parent resumed.

If P1 fails its parent also fails.

c - In fact, a process is not systematically added to the process tree. It is first put in a process buffer and later moved to the process tree only if it suspends or forks. In the former case its parent is resumed and in the latter case its offspring will be put in the process buffer. This buffer avoids the creation of processes that will be immediately removed from the process tree and allows to keep only the minimum information about the new virtual process.

SPM is an abstract machine (just as the WAM-Warren Abstract Machine for Prolog) with a set of instructions which manipulate the data structures constructed by the computational model. The idea here is to put the whole set of SPM instructions into the μSyC's micro-control store (Fig. IV.1)

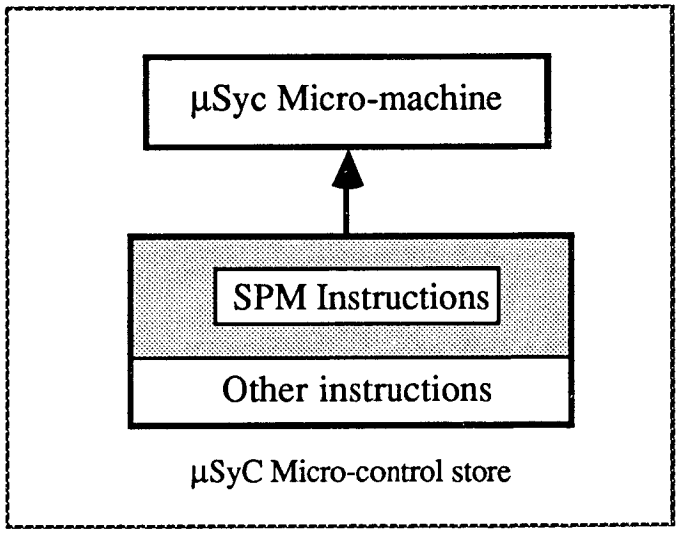


Fig IV.1

The structures inherent to the SPM machine are discussed in the next section.

IV.1 Mapping of the SPM structures on μSyC

a) The process tree

The process tree is a binary tree whose nodes are either process descriptors or query descriptors (Fig IV.2) :

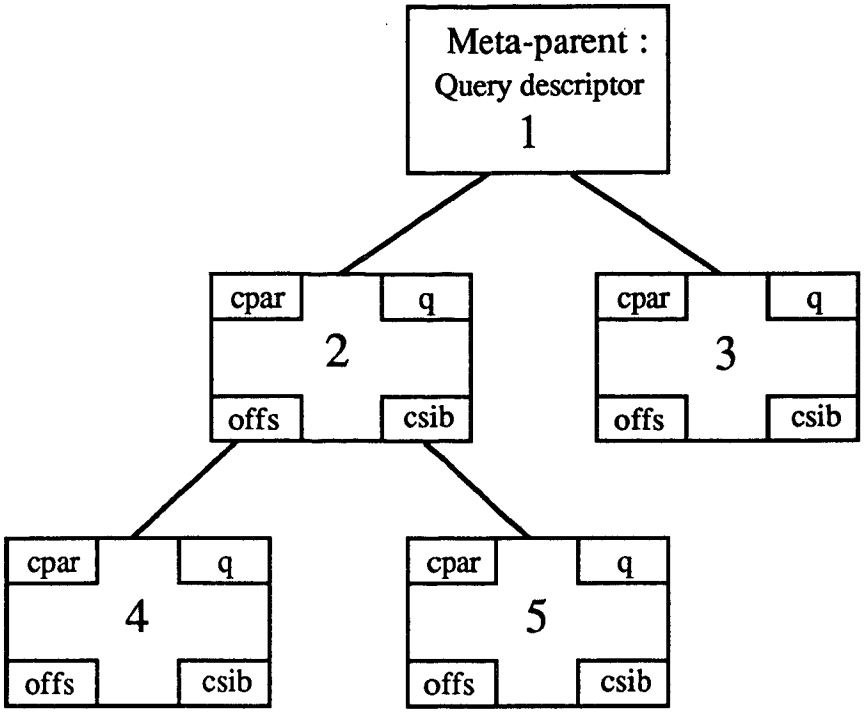


Fig. IV-2

As represented above, a node consists of several pointers :

cpar : points to the parent process
offs : points to the first offspring process
csib : points to the first sibling process
q : to the metaparent process

There are two other variables not depicted in the schema :

wvar : points to the variable the process is waiting for

wsib : points to next process waiting on the same variable

Depending on the kind of the node, a descriptor has additional fields :

For a process descriptor :

psta : process status (RUNNING, RUNNABLE, WAIT, SUSPENDED)

atyp : argument vector type (OWN, INHerited)

p : process code pointer

a : argument vector

For a query descriptor :

runn_h : points to the first runnable process attached to the query

runn_t : points to the last runnable process

ilimit : inference limit

status : points to the query status

next : points to the next query descriptor

It is worth noticing that each query has a separate runnable list to control the amount of inferences allocated for a query. Figure IV. 3 below shows a typical SPM query list :

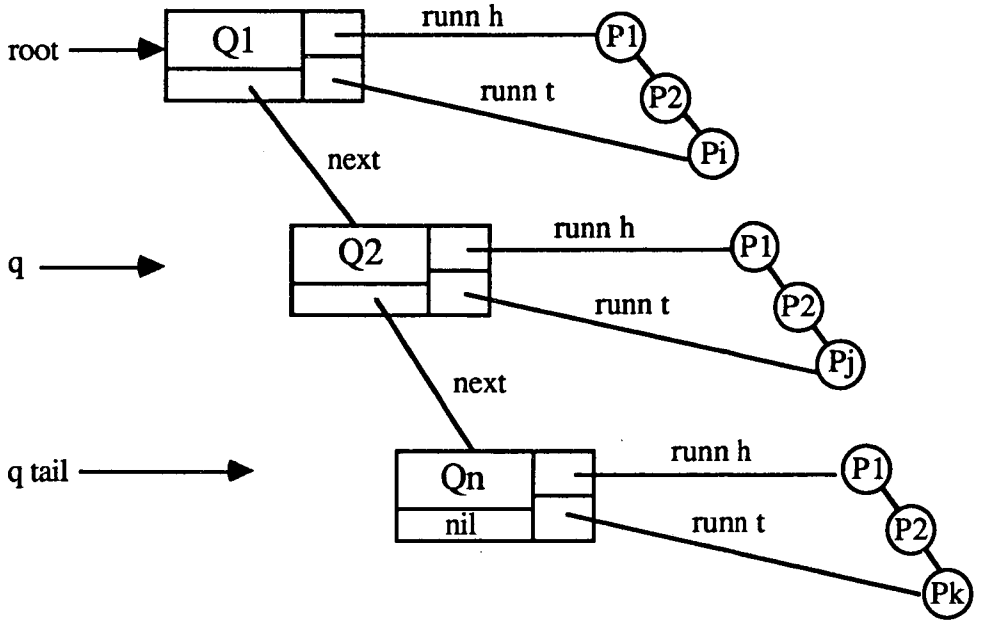


Fig. IV.3

During its execution SPM manipulates various kinds of information located in 3 areas :

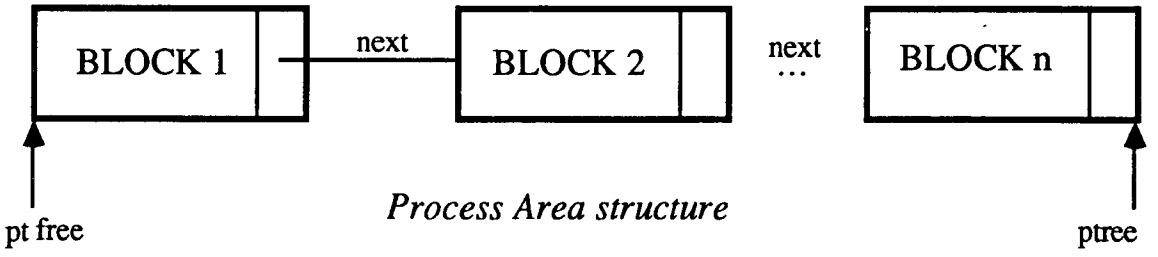
b) The code area: this contains the SPM code

c) The Process area: it is split into dynamically allocated fixed size blocks which hold descriptors and argument vectors.

Two pointers are associated with this area :

- **ptree** : points to the top of the process area
- **pt_free** : points to the first free block in the process area

To run the SPM naive reverse benchmark we have assumed that the process has the following structure :

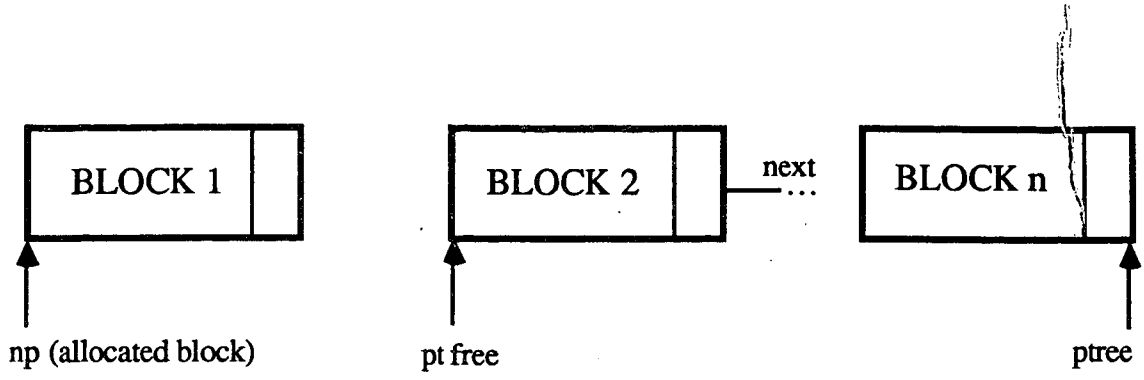


Process Area structure

During the initialization all the blocks are sequentially chained, pt_free points to the first block and ptree to the end of the process area.

Allocation mechanism :

Each time a new block is requested, a pointer to the first block of the free list is returned and pt_free is set to the address of the next free block. If pt_free equals ptree no address is returned but an error is generated.



Process Area after : ALLOCA np

De-allocation mechanism :

There is no need for garbage collection of process blocks since de-allocation is explicitly requested by SPM when a process dies. During the call of the de-allocation routine the address of the block to be de-allocated is provided.

Each time a block is de-allocated, it is added to the head of the free list and pt_free is set to its address.

Pointers pt_free and ptree are put into two internal µSyC registers to accelerate the allocation and de-allocation mechanisms.

d) The heap

This third area contains all the variables and other data structures manipulated by the processes. It is composed of fixed size cells. For µSyC we will assume that each cell is 32 bit wide. A cell is itself divided into two fields: the tag and the value. The following are the different kinds of tags and values that we can find in SPM:

TAG	VALUE
REF	pointer to a cell
UNB	unbound variable
CON	pointer to a dictionary entry
INT	integer
NIL	NIL pointer
LST	pointer to a list
STR	pointer to a structure

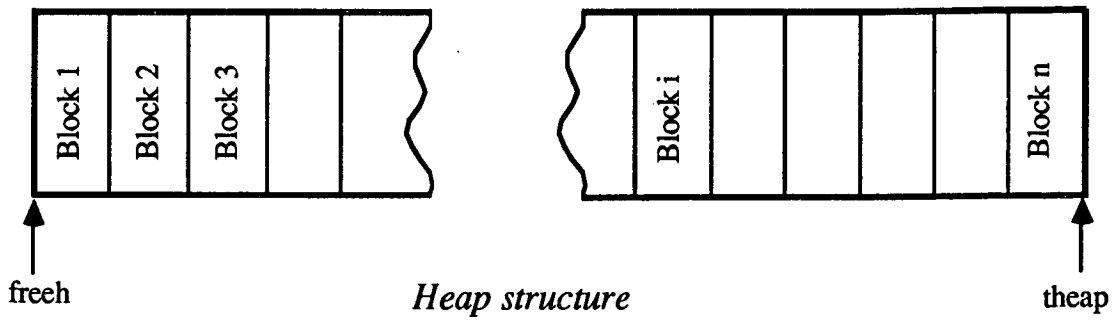
In µSyC, the tag occupies the 4 most significant bits of the cell and the remaining 28 bits are used to hold the cell value.

As explained in section II, tagging, untagging and tag checking operations have been optimized in µSyC by implementing them at the hardware level.

Cells allocation :

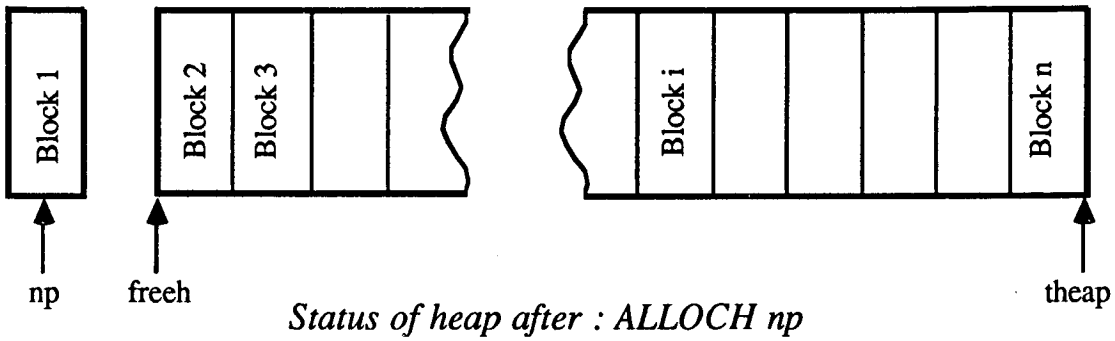
To run the naivereverse benchmark we will choose a simple cell allocator which only requires two pointers:

theap : pointer to the top of the heap
freeh : pointer to first free cell in the heap



Heap structure

When the cell allocator is called, the cell pointed to by freeh is returned and freeh is incremented



Status of heap after : ALLOCH np

Cells de-allocation :

Unlike process blocks, cells are not explicitly freed by SPM as processes die. So when the freeh pointer meets the theap pointer (adress to top of the heap) the allocator is not able to allocate more cells and garbage collection is then needed to collect the unused cells. The garbage collector proceeds as follows :

- Accessible cells are located and marked by recursively accessing the process tree nodes.
- The accessible cells are then packed to the bottom of the heap and the freeh is updated.

Garbage collection has not been implemented yet although the possibility of tagging cells and automatically testing them in μSyC should provide for a very efficient garbage collector.

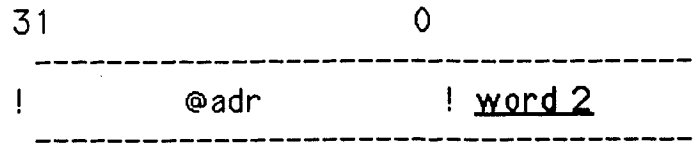
IV.2 Micro-programming the SPM macro-instructions

Each SPM macro-instruction is normally one 32 bit word wide (opcode + arguments). Only those instructions which need an address or an integer constant as an argument are two 32 bit words wide (an additional 32 bit word is used only for the address or the constant).

The word containing the opcode is structured as follows :

```
31      23      15   7    0
 ----------------------------------
! opcode ! arg1 ! arg2 ! arg3 ! word 1
 ----------------------------------
```

The second word (when needed) simply contains a 32 bit address or a 32 bit constant :



To each SPM instruction corresponds a μSyc program that implements it. This micro program varies from 50 to 150 lines of code, or the equivalent of 200 to 600 μSyc instruction for each SPM instruction. Currently the 30 major SPM instructions are implemented.

V SPM BENCHMARKS

V.1 Description of benchmarks

Three benchmarks have been run on μSyC :

1) The naive-reverse :

Parlog code of naive-reverse :

```
mode rev(?,^).
rev([X|Xs],Y) <- rev(Xs,Ys), append(Ys,[X], Y).
rev([], []).

mode append(?,?,^).
append([X|Xs], Ys, [X|Zs]) <- append(Xs, Ys, Zs).
append([], Ys, Ys).
```

2) The quick sort :

Parlog code for the quick sort:

```
mode qsort(?, ^, ?).
qsort([X|Xs], S1, L2) <-
        part(X, Xs, S, L), qsort(S, S1, [X|L1]), qsort(L, L1, L2).
qsort([], X, X).

mode part(?, ?, ^, ^).
part(X, [Y|Ys], [Y|S], L) <- X >= Y | part(X, Ys, S, L).
part(X, [Y|Ys], S, [Y|L]) <- X < Y | part(X, Ys, S, L).
part(X, [], [], []).
```

3) The Takeushi benchmark

Parlog code for the takeushi benchmark:

```
mode tak(?, ?, ?, ^).
tak(X, Y, Z ,A) <-  X>Y |
                    X1 = X - 1, Y1= Y - 1, Z1 = Z - 1,
                    tak(X1, Y, Z, A1),
                    tak(Y1, Z, X, A2),
                    tak(Z1, X, Y, A3),
                    tak(A1, A2, A3, A).
tak(X, Y, Z, Z) <- X<=Y | true.
```

V.2 Environment for running SPM benchmarks

A μSyC emulator has been written in C on a BULL UNIX machine (SPS9) not only for debugging the entire micro-code for the SPM but also to run the benchmarks we have mentioned in the previous paragraph.

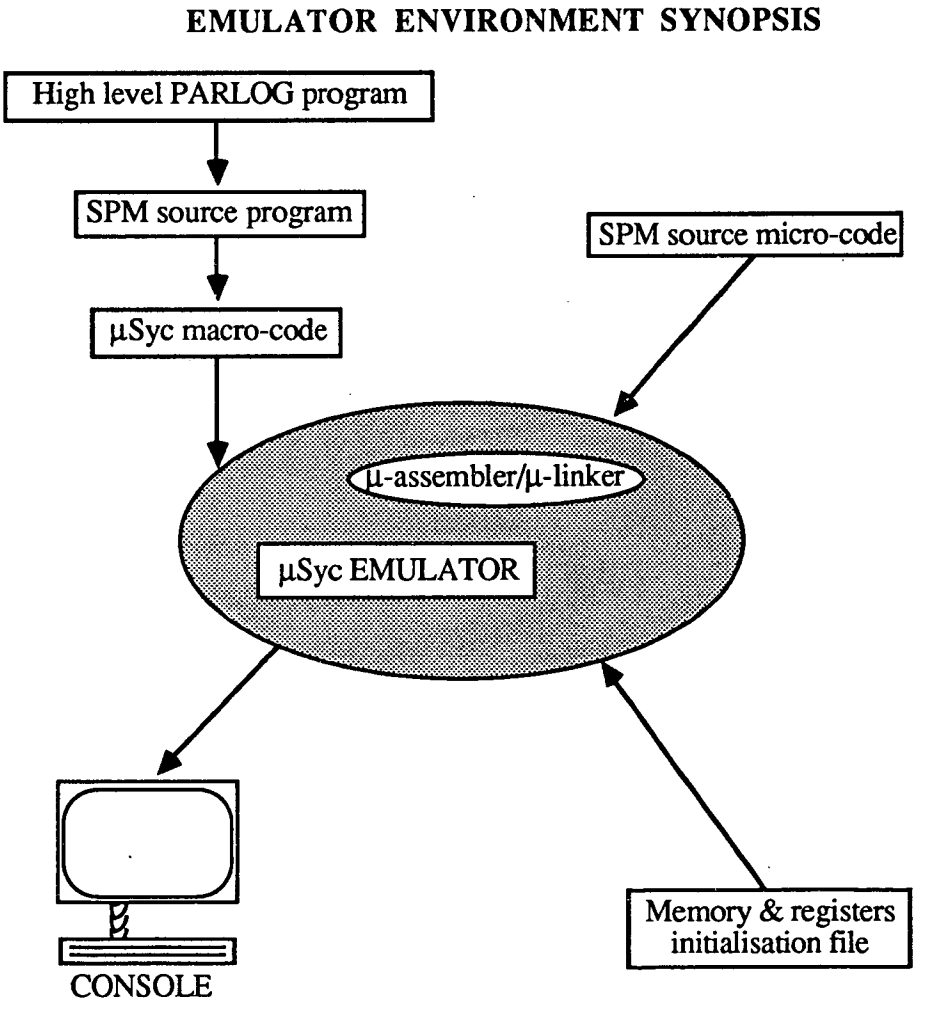


Fig. V. 1

The emulator has several inputs: (Fig. V. 1)

- An SPM program in μSyC code format (see Annexe A2). It is obtained by assembling an SPM program, itself resulting from a compilation of a high level PARLOG program (see paragraph V.1).
- The SPM micro-code source which is first micro-assembled by a built-in micro-assembler and translated into a compact intermediate code used by the emulator. The micro-code can be distributed among several files. Each file is then micro-assembled and dynamically linked. If one or more micro-assembly errors are encountered, the erroneous micro-program has to be modified and then reloaded. This can be done by invoking an external editor within the emulator.
- A configuration file which is used to initialise the internal μSyC registers, the working memory (which contains the SPM code, the heap and the process area) and the memory access time.

Emulator output :

- The emulator provides the number of inferences required for an SPM program and the number of micro-cycles spent to run it. The number of micro-cycles is strictly exact as it includes the possible wait states to access central memory (memory access time is one parameter of the emulator).

Brief presentation of the emulator capabilities

The μSyC emulator provides an invaluable amount of debugging facilities :

- An unlimited number of micro-breakpoints can be set and reset.
- Registers and external memory can be patched, either manually or from a UNIX file.
- Registers and external memory contents can be traced at each micro-cycle.
- The micro-instruction being executed can also be displayed at each cycle (it is previously micro assembled).
- Micro-code can also be executed step-by-step, and alternate between running and step-by-step mode.
- Micro-code execution can be stopped at any time by pressing a key. This option is intended to stop programs which may infinitely loop or to examine the machine state or the memory at any moment in time.
- An external editor can be invoked to modify a file.
- Several commands accessible from the emulator menu can be included in a batch file which can then be executed by the emulator.

V.3 Results

The following table makes a comparison between the results obtained for μSyC and those provided by Imperial College for a SEQUENT machine, which is a multi-processor machine based on 24 INTEL 80386 micro-processors and a SUN 3/75.

	naive-reverse (400 elements)	QSORT (400 elements)	Takeushi(18,12,6,X)
SEQUENT (1x80386)	15.010	6.363	4.028
SEQUENT (10x80386)	116.814	42.809	36.348
SUN 3/75	14.5	5.6	3.4
μSyC	64.5	35.38	27.876

Benchmark results in KLIPS

For μSyC, we have assumed that memory can be accessed within 2 micro-cycles, which means that memory access time must be less than 150 ns.

VI Conclusion

We have described in this paper an implementation of the Sequential Parlog Machine in firmware on the microprogrammable symbolic coprocessor μSyc. Although garbage collection is not yet microprogrammed, the performance figures obtained for classical benchmarks seem very encouraging and compare favorably to those of a ~5 processor Sequent implementation.

There are four main reasons for which μSyC has good performances:

- It has an important register set (44, 32 bit registers)
- It implements fast tag checking
- I/O is optimized
- It benefits from the parallelism within the micro-machine
- Finally the compilation process is optimised since the levels of interpretation are minimized (SPM is in Firmware)

Future work will study the implementations of more efficient intermediate Parlog machines on a shared memory multiprocessor machine supporting μSyc.

REFERENCES

[Cla 85] Clark K.L. and Gregory S. : "PARLOG: parallel programming in logic". ACM Trans on Programming Languages and systems, 1985.

[Cou 87] Couprie M., Garcia J., Maréchal T., Terral D.: "μSyC : Coprocesseur Microprogrammable pour les Applications Symboliques". Journées Firftech Bases de Données et Intelligence Artificielle, Paris, apr. 87.

[Fos 86] Foster I.T., Gregory S., Ringwood G.A and Satoh K. : "A sequential implementationof PARLOG". In Proc. of the 3rd International Logic Programming Conference. (London, July), E. Shapiro (Ed), New York, Springer-Verlag, pp. 149-156, 1986.

[Gon 84] Gonzalez-Rubio R., Rohmer J., Terral D.: "The SCHUSS Filter : A Processor for Non-Numerical Data Processing". 11th Annual International Symposium on Computer Architecture, Ann Arbor, 1984.

[Gon 86] Gonzalez-Rubio R., Bradier A., Rohmer J.: "DDC Delta Driven Computer. A Parallel Machine for Symbolic Processing". ESPRIT Summer School on Future Parallel Computers, University of Pisa, june 1986.

[Gre 87a] Steve Gregory, Ian Foster, Alastair D. Burt and Graem A. Ringwood : "An abstract machine for the implementation of PARLOG on Uniprocessors", january 1987.

[Gre 87b] Gregory S. : Parallel Logic Programming in PARLOG. Reading, Mass.: Addison-Wesley, 1987.

[Sha 87] Shapiro E. : Concurrent Prolog. Collected papers, Ehud Shapiro (ed), MIT Press series - Logic Programming 1987.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique